AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN SYSTEM

THESIS

Timothy Martin Jacobs
Captain, USAF

AFIT/GCS/ENG/91D-09

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
Timothy M. Jacobs, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-09

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
RL/OCTS
Rome Labs
Griffis AFB, NY 13441

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This thesis attempts to prove that the commercially available *ObjectStore* data management system provides the performance and functionality necessary to support a complex engineering design system. This is accomplished by modifying the *Magic* VLSI circuit layout design system to eliminate its current Unix file data management system and replace it with ObjectStore. The approach to this research effort includes a design recovery of the Magic system and identification of its key data management functions. These functions are then modified to take advantage of the database management facilities of ObjectStore. Additional code is added to instrument performance measurement of both the original and the ObjectStore versions of the Magic system. Testing is accomplished using existing Magic commands to test key database performance criteria. The ObjectStore version of Magic performed better than the original version for some performance criteria and significantly slower than the original version for other criteria. The conversion effort was difficult and time consuming due to the complexity of the original Magic software and the ObjectStore database management system. A more specific implementation of ObjectStore capabilities is necessary for conclusive results.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Oriented, Database Management System, Computer-Aided Design, Very Large Scale Integration	72
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT/GCS/ENG/91D-09

AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Timothy Martin Jacobs, B.S., M.S.B.A.

Captain, USAF

December 1991

Table of Contents

List of Figures

List of Tables

Abstract

Despite the many advantages provided by database management systems, many complex applications spurn their use in favor of application unique file management systems. This is primarily due to the inadequate performance of conventional database systems. Recent research, however, has indicated the potential for object-oriented database systems to fulfill the performance requirements which these complex applications demand.

Among these complex applications are engineering design systems. This thesis attempts to prove that the commercially available *ObjectStore* data management system provides the performance and functionality necessary to support a complex engineering design system. The *Magic* circuit layout design system is modified to eliminate its current Unix file data management system and replace it with ObjectStore.

The approach to this research effort includes a design recovery of the Magic system and identification of its key data management functions. These functions are then modified to take advantage of the database management facilities of ObjectStore. Additional code is added to instrument performance measurement of both the original and the ObjectStore versions of the Magic system. Testing is accomplished using existing Magic commands to test key database performance criteria.

The ObjectStore version of Magic performed better than the original version for some performance criteria and significantly slower than the original version for other criteria. The conversion effort was difficult and time consuming due to the complexity of the original Magic software and the ObjectStore database management system. A more specific implementation of ObjectStore facilities is necessary for conclusive results.

AN OBJECT-ORIENTED DATABASE IMPLEMENTATION OF THE MAGIC VLSI LAYOUT DESIGN SYSTEM

I. Introduction

1.1 Overview

Database management systems (DBMS) have proven themselves in a large variety of computer applications. Today's commercial DBMSs provide an effective tool for managing large repositories of data while providing access to multiple users and applications. All users have access to the same data since it is stored in a single location. Concurrency control and recovery methods ensure data consistency despite multiple users and hardware or software failures. Applications can be easily added without knowledge of the physical layout of the data. Overall, modern DBMSs provide numerous advantages over alternate data management facilities.

Despite these many advantages, there are numerous computer applications which continue to spurn the use of a DBMS in favor of their own unique application file system. Among these are engineering design systems which are heavily dependent on large amounts of computer data. Few applications systems which support these design processes are integrated with a DBMS.

Although conventional databases are unable to adequately support these applications, the need for some sort of database support becomes evident as the systems proliferate among more powerful workstations and in increasingly complex engineering environments. Object-oriented database management systems (OODBMS), while still not widely available, have shown the potential for providing the necessary database support for these complex, data intensive applications.

This thesis examines the potential of object-oriented databases to support complex design applications. A very large scale integrated (VLSI) circuit design tool is implemented on a newly released commercially available OODBMS. The performance of this tool as

implemented on the OODBMS is compared to its performance as currently implemented with a flat file system. Conclusions are drawn about whether an OODBMS can adequately support a complex, data intensive, automated design system.

1.2 Background

Computer-aided engineering design tools have been developed for a number of applications such as integrated circuit design, software engineering, and machinery design. Typically, such tools assist the designer by providing graphical representations of the desired object along with a narrative or symbolic description. A repository of previously defined objects exists from which the designer may choose an object to incorporate into the new design. A designer can start with a high level view of the desired object and gradually refine the design down to the tiniest detail. This process may occur over a period ranging from hours to months.

The data associated with a typical design contains many complicated relationships among various data items. A complete design object, such as a VLSI circuit or an automobile, may have thousands of individual data components. Additionally, computer representation of graphical and textual data requires vast amounts of storage and is often unique to a specific hardware or software system.

The relational database management systems which are currently widely available were originally designed to support business applications. As such, these DBMSs are oriented toward individual data records and simple data objects such as bank accounts. The relationships among various objects are limited and rather well defined. Data types are typically numbers or short textual descriptions. Individual transactions usually involve only a few operations on a small number of records and relations. Such transactions are completed in fractions of a second.

The difference between the data requirements of engineering design tools and those of business applications for which currently available database systems were developed is significant. Attempts at implementing design tools on these conventional DBMSs have produced results ranging from excessive processing time to outright failure. To overcome

these problems, considerable research has been undertaken to develop a DBMS suitable for the data representations and time constraints of complex applications such as engineering design tools. Considerable promise in this area has been shown using object-oriented database management systems.

The theory behind object-oriented database management systems is to incorporate the object-oriented programming paradigm into a database system which provides concurrency control, failure recovery, relationship modeling, and data persistency. Key elements of the object-oriented paradigm, which simplify representation of complex applications, include object identity, data encapsulation, complex state, and inheritance. Many experimental OODBMSs, and the few commercially available OODBMSs, also support complex data types, multiple versions, and long transactions.

Table 1 presents the characteristics of an engineering design tool along with a comparison of support provided by currently available relational DBMSs and object-oriented DBMSs. From this table it is evident that a good OODBMS can potentially provide the database support necessary for a complex application such as an engineering design tool.

1.3 Problem Statement

Traditional database management systems (typically relational models) perform too slowly for complex, data intensive applications such as computer-aided engineering design. As a consequence of this slow performance, most design tools have their own file system. Such file systems require data management to be accomplished manually, often by various individuals in the design organization. Manual data management, in turn, increases the potential for errors such as deleting or modifying the wrong version of a design. In addition, a change to the structure of any of the files requires changes to all programs which reference the modified structure. This increases the maintenance effort required for the system.

1.4 Research Objectives

The primary purpose of this thesis is to determine whether or not an object-oriented database management system provides the performance and functionality necessary to

Characteristic	Design Tool Example	Traditional DBMS	Object-Oriented DBMS
Complex State	References to subcomponents within circuit.	A key is required for each sub-component. Joins are required to merge into a single object.	Fundamental to the object-oriented paradigm.
Inheritance	New adder inherits attributes of a typical adder and modifies them to fit a particular circuit.	Complete specification of the schema must be defined *a priori*.	Fundamental to the object-oriented paradigm.
Complex Data Types	Graphical representation of a circuit.	Only supports basic data types such as integer and character.	Supports graphical and textual data and allows user to define data types.
Multiple Views	Top level view of design or more detailed look at a sub-component.	Must be defined in the application. Limited by record oriented retrieval.	Can be specified as a method for the object. Data is more easily retrieved using object-oriented storage techniques.
Multiple Versions	Current and historical versions.	May support multiple versions of individual records.	Generally built in as a tree structure with root node representing a version. Tree includes all objects which make up the version.
Phased Development	Top down design.	Not supported. Entire schema must be defined *a priori*.	Refined schema can inherit characteristics of a higher level and modify for next phase.
Large Data Volume	Thousands of sub-components in a circuit.	Limited only by physical storage; however, record-oriented storage may limit the size of record, causing multiple record retrievals for a single object.	Clustering by object reduces the number of disk accesses. Complex data types remove object size restrictions.
Long Transaction Duration	Designer takes two weeks to modify a specific circuit design.	Built around short business transactions. Inefficiency and failure occur with long transactions.	More appropriate concurrency control and failure recovery methods used to support long transactions.
Fast Performance	Thousands of sub-components are retrieved and displayed in seconds.	A single view requires multiple joins and many individual accesses.	Designed to retrieve large amounts of data at once.

Table 1. DBMS Support of Engineering Design Tool Characteristics

support a typical computer-aided design tool. To fulfill this objective, the design tool must maintain all functionality provided by the existing file management system. Performance of the design tool must remain acceptable to the tool users. This is specified as no more than a ten per cent increase in response time over the current implementation.

This thesis should also demonstrate that conversion of a design tool application from a unique flat file management system to an OODBMS is not a difficult or time intensive task. To meet this goal, the time and effort spent converting to the OODBMS must be cost effective with respect to the increased utility of a database management system and the reduction in future software maintenance costs.

1.5 Assumptions

If the results of this thesis are to be applied to other engineering design tools, the tool implemented (i.e. the *Magic* VLSI circuit design tool) must be typical of all engineering design tools.

Due to the inexactness of the available techniques for measuring results, assumptions are made about disk access times and actual processing times. These assumptions are described and justified along with the results obtained.

A similar lack of preciseness and a large amount of subjectivity occur when calculating the added value of a DBMS and when estimating future maintenance costs. Assumptions regarding these values will also be described and justified along with the results.

1.6 Scope and Limitations

This thesis implements a direct conversion of code for the Magic VLSI circuit design tool from the existing file system to the *ObjectStore* object-oriented database management system. This includes conversion of *C* code for compatibility with the ObjectStore *C++* compiler. Software redesign is accomplished only as necessary to make Magic work with ObjectStore.

No attempt is made to improve the performance of Magic by modifying the existing software structure. All algorithms and data structures remain unchanged. Data structures are added only to replace current code which directly accesses physical data files.

1.7 Methodology

Magic is a VLSI circuit layout design tool which is commonly used at the Air Force Institute of Technology (AFIT) and other U.S. institutions. A number of people at AFIT are familiar with the functionality and performance of this tool. It contains the characteristics of a typical computer-aided engineering design system as described in Section 1.2. Magic manages data with its own unique flat file system. Because of its familiarity among AFIT personnel, its typical design tool characteristics, and its current flat file management system, Magic is selected to test the potential of an object-oriented database management system.

Few commercially available OODBMSs currently exist. Among these is the ObjectStore database management system developed by *Object Design Incorporated* of Burlington, Massachusetts. This OODBMS has been made available to AFIT and is used for this thesis to implement the Magic VLSI circuit design tool.

The first part of this thesis requires implementation of Magic with the ObjectStore DBMS. This implementation follows the steps described below.

1. Accomplish a design recovery on the Magic system. This provides an understanding of the software structure and functionality and is necessary since a limited amount of design documentation currently exists.
2. Replace existing file input and output with persistent database structures. This involves eliminating all file input and output and making program data structures persistent. If no data structure exists because data is directly extracted or replaced in a file, then a persistent data structure is created which is similar to the file structure.
3. If necessary, redesign software which can not be implemented with ObjectStore. Redesign is accomplished in a manner similar to existing Magic code with emphasis on generally accepted object-oriented design procedures.

Performance comparison of the existing implementation of Magic to its implementation using ObjectStore requires computation of performance characteristics. These calculations involve the tasks described here.

1. Instrument code in critical regions of the software to measure processing time. This code must occur in the same place for both the existing implementation of Magic and the ObjectStore implementation. In general, those sections of software most likely affected by the database management system are instrumented for processing time measurement.
2. Instrument code in the existing software to measure disk accesses. Accomplish this for those segments of code involving file input and output.
3. Extract disk access information from the ObjectStore DBMS. This extracted information involves the same data as the file access data in item 2 above.

The second objective of this thesis is to show that the conversion effort is cost effective with respect to reduced future maintenance and increased utility of a DBMS. Measurements for these comparisons are reached with the following steps.

1. Identify time spent converting the existing Magic system to an OODBMS.
2. Subjectively evaluate the difficulty of the conversion based on the learning curve for ObjectStore and the actual code modifications which were accomplished.
3. Subjectively estimate the benefits of having Magic implemented with a DBMS. This is accomplished by interviewing users of Magic and by researching other organizations' uses of design tools. The results of the interviews and research are interpreted as to their compatibility or incompatibility with ι DBMS.
4. Subjectively estimate the reduction of future maintenance costs based on existing studies of the cost of maintenance and consultations with software personnel who have experience in software maintenance.

1.8 *Materials and Equipment*

This research effort utilizes the ObjectStore (version 1.1) database and development facilities on a *Sun Sparc II* workstation. ObjectStore provides all tools necessary to convert the Magic system. The Magic software is also available on a *Sun Sparc II* workstation for modification and testing.

1.9 *Document Summary*

Chapter 2 describes previous research on object-oriented databases in support of computer-aided design applications. This chapter also describes the functional aspects of the Magic VLSI layout system and the ObjectStore DBMS. Chapter 3 presents a design recovery of Magic and discusses the methodology employed in implementing Magic with the ObjectStore database. A comparison of Magic performance and the effort involved in conversion to ObjectStore is contained in Chapter 4. Chapter 5 includes conclusions reached regarding the objectives of this thesis and recommendations for further research.

II. Databases and Design Systems: Relevant Issues

2.1 Overview

The field of object-oriented databases is relatively new and few implementations have actually been put into practical use. This chapter reviews existing research on object-oriented databases with engineering design systems and discusses the potential benefits. Additional database design issues are discussed along with their impact on performance. A brief overview is presented of the Magic layout design system and the ObjectStore database management system.

2.2 Engineering Design and Database Management Systems

Developers of automated design systems have long been searching for database management systems which meet the performance requirements necessary for manipulating complex engineering entities. Thomas Sidle identified *Weaknesses of Commercial Data Base Management Systems in Engineering Applications* (17) as early as 1980. These weaknesses include slow response, excessive discipline imposed on the software development activity, difficulty of satisfying engineering requirements, and organizational problems associated with database support. The primary reason for these inadequacies is the orientation of existing DBMSs toward business applications.

A typical business database consists of a large number of structurally simple records. Most transactions involve simple requests to locate and perform simple operations on a small number of records. The record structures, operations, concurrency control techniques, and failure recovery methods of conventional DBMSs are designed to support these business databases. Inefficiencies and failure result when these DBMSs are used to support engineering design applications. (17)

Instead of conventional DBMSs, many database experts have proposed Object-Oriented Database Management Systems as more appropriate for engineering design systems. According to Sandra Heiler *et al* (4), an object-oriented approach to data management supports engineering design requirements by allowing users to define relationships among engineering objects and by providing facilities for defining complex objects and

version configurations. Changes to data items are controlled by limiting the operations that can be applied to an object to those which are specified in the object type description. By allowing the user to specify the behavior of an object, OODBMSs support triggering of changes to derived objects. Operations can also be specified for logging changes to an object and for defining policies for relationships between objects. The object-oriented paradigm provides a better model for mapping to the mental model of the users. System maintenance is simplified since "changes to the implementation of one object or object type will not require changes to others (4:339)." Using the inheritance characteristic of object-oriented DBMSs, new objects can be added or old objects modified as requirements are refined. Different implementations of an operation or data structure can be defined without affecting the interface of the object to other objects.

Rajiv Gupta *et al* (3) point out similar advantages of an OODBMS. The object-oriented paradigm mimics real-world objects, encourages gradual evolution of a design, and encourages code reusability. By storing powerful data structures persistently on disk the need for local memory-resident structures is eliminated and retrieval can be optimized by caching entire objects in memory. Aggregation allows representation of complex objects which reference a number of constituent objects. An object which is a specialization of another object can be modeled such that it inherits the characteristics of the higher-level object. A group of objects can be represented as a class such that each instance of the class has the same attributes and operations. Gupta *et al* also point out a few drawbacks of OODBMSs. These include increased use of disk space, slower response than file-based CAD systems, and difficulty in finding errors hidden in an abstracted implementation layer of an object.

2.3 Database Performance

If database management systems are to replace the unique file management systems which are common in engineering design applications, the performance of these DBMSs must be comparable to the existing systems. The following sections discuss methods of evaluating database performance and some design issues which may affect the performance of object-oriented database systems.

2.3.1 Performance Evaluation. Much of the literature on database performance evaluation addresses the results of standard benchmarks as applied to various DBMSs. While most of these benchmarks reflect typical applications for relational DBMSs, R.G.G. Cattel has developed an approach for measuring the performance of object-oriented database systems (2). Before discussing his approach, however, Cattel points out that "The most accurate measure of performance for engineering applications would be to run an actual application, representing the data in the manner best suited to each potential DBMS (2:364)."

Cattel proposes a database of parts on a circuit board with connections between them. He summarizes the three most important measures of performance in an object-oriented DBMS as:

Lookup and Retrieval. Look up and retrieve an object given its identifier.

Traversal. Find all objects in the hierarchy of a selected object.

Insert. Insert objects and their relationships to other objects.

To meet the performance requirements of engineering applications, Cattel suggests that a DBMS must be able to perform 1000 random operations per second. He noted that none of the OODBMS implementations in research or production environments met his criteria when his paper was written in 1987.

Berre and Anderson's *HyperModel* benchmark (1) presents a similar approach to performance measurement. In addition to the operations proposed by Cattel, the HyperModel benchmark includes:

Sequential Scan. Visit each object in the database sequentially.

Closure Operations. Perform operations on all objects reachable by a certain relationship from a specified object.

Open-and-Close. Time to open and close the database.

2.3.2 Design Issues Impacting Performance. One of the key design issues affecting OODBMS performance is whether or not to cluster subobjects and their referencing objects in physical storage. Jhingran and Stonebraker (6) address this issue and the effect of caching. They run a series of experiments using a relational DBMS and a hierarchical data structure in which the number of shared superobjects and subobjects is varied. Jhingran and Stonebraker show clustering to be advantageous only when sharing of subobjects is relatively low. When retrieving objects in a homogeneous collection, clustering decreases performance since the objects are stored with their superobjects and are no longer contiguous on disk. Caching, on the other hand, is generally viable except when a large number of updates is made.

Another design issue affecting performance is the use of indexing. Kim *et al* (7) evaluate two different indexing techniques. The first of these is an index maintained on an attribute of a single class. Kim *et al* identify this as a *single-class index*. A *class-hierarchy index* is maintained on an attribute of all classes in a class hierarchy rooted at a particular class. Two different cases were studied with the same set of range values varied for each case. In one case all of the range key values are assumed to be in any one of the classes in the hierarchy. The second case scatters the range values evenly in two classes. In both cases *class-hierarchy indexing* generally results in fewer index pages being fetched for a given query if there are at least two classes in a class hierarchy.

2.4 The ObjectStore Database Management System

Because of the complexity of object-oriented databases and the immaturity of the field, few commercially available object-oriented databases exist. Of those that do exist, many are merely object-oriented interfaces to a relational database. Only recently have any databases been commercially released with memory management techniques suitable for object-oriented database management. One such database management system is ObjectStore, an object-oriented database management system released in 1990, with an upgrade release in 1991, by Object Design, Incorporated.

ObjectStore supports most of the object-oriented database characteristics listed in Table 1. The database design language is *C++* which provides support for complex state,

inheritance, and user defined data types (18). Specific views for an object can be expressed in the *C++* functions associated with that object. ObjectStore also provides a versioning mechanism to support long transactions and multiple versions. To handle large amounts of data, ObjectStore uses a memory mapping and page swapping mechanism which can be customized by the database designer. (13) ObjectStore does not support schema evolution. Any change to a schema makes data created with the old schema unreadable by the new program. (16)

In addition to its uniquely object-oriented characteristics, ObjectStore also has traditional database management facilities. All access to the database must occur within a transaction. All data manipulation which occurs within a transaction is not visible outside of the transaction until the transaction is complete. This avoids the potential data integrity problems which can occur if two separate applications modify or use the same piece of data simultaneously. Data integrity is also ensured through relations and inverse relations which synchronize related objects when one of them is modified. ObjectStore provides tools for managing collections (groups of homogeneous data) and supports query processing over these collections. (13)

ObjectStore's *Virtual Memory Mapping Architecture* (VMMA) is key to its performance. This architecture allows persistent data stored in ObjectStore to be handled in the same way as non-persistent (transient) data. Large amounts of data can be retrieved and manipulated with minimal overhead through virtual memory management. When referenced data is not in main memory, a page fault occurs which is intercepted by ObjectStore so that it can retrieve the data from the database into memory. The overall effect of the memory mapping architecture is to provide the developer a single view of memory — basically expanding the program memory to the size of the database. (13)

For an application to work with ObjectStore three auxiliary processes are required — the *ObjectStore Server*, the *Directory Manager*, and the *Cache Manager*. The Server handles all storage and retrieval of persistent data. The Directory Manager manages ObjectStore directories much as Unix manages its directories. The Cache Manager controls swapping of data between the cache memory associated with an application and the virtual database memory. (13)

ObjectStore provides interfaces to both *C* and *C++*. It also has its own Data Definition and Manipulation Language (DML) which is a superset of *C++*. The DML simplifies *C++* library routines, such as setting the database root and controlling transactions, by replacing a sequence of *C++* commands with a single DML command.

2.5 The Magic VLSI Layout Design System

Due to the complexity and cost associated with the design and creation of VLSI circuits, computer-assisted tools are essential. One of the key steps in the circuit development process is design of a physical layout of the circuit which can be directly implemented on a chip. Tools for manipulating and verifying this design are necessary to keep track of the numerous components and connections and to minimize the risk of the chip failing after it has been manufactured. One such tool is the Magic VLSI layout system which was originally developed at the University of California in Berkeley with the latest release from Digital Equipment Corporation's Western Research Laboratory in 1990.

The purpose of Magic is to increase the power and flexibility of previous layout editors so that designs can be entered quickly and modified easily. (15) To accomplish this goal, Magic provides basic commands for creating, copying, modifying, and deleting circuit components along with capabilities for automated circuit routing and continuous checking of design rules. Circuits can be created completely from scratch or through hierarchical inclusion of any number of sub-components. File extraction tools have also been included as part of Magic for compatibility with circuit testing and manufacturing systems. (10)

To improve performance and simplify the designer's view of a circuit, Magic implements some unique features. The geometrical contents of a circuit are represented using a technique called *corner stitching*. In this technique, a circuit contains a number of corner-stitched planes, each of which consists of a number of rectangular tiles representing the physical material to be included in the actual circuit. These tiles are the fundamental data units represented in the database. Each tile is linked in its lower-left corner to those tiles to its left and bottom. Another link in the upper right corner connects the tiles to the right and top. Figure 1 demonstrates how three filled in tiles (enclosed with solid lines) would be stitched together with blank tiles (dashed lines) in a single plane. Corner stitching permits

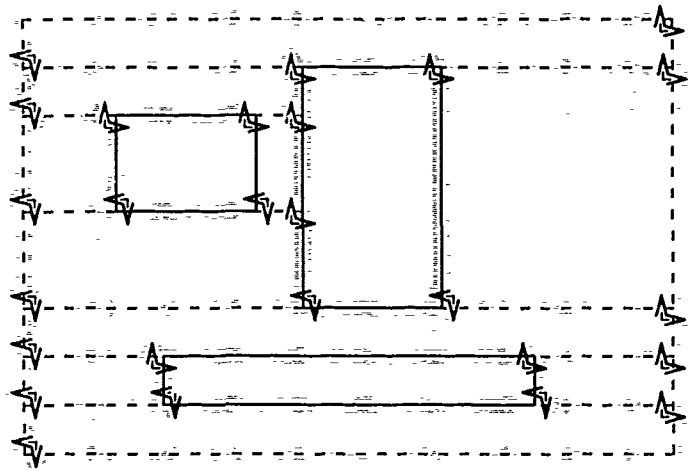

Figure 1. Corner stitching of tiles in a plane (15)

search operations to be performed more efficiently (15); however, iterating through all tiles in a cell may require traversing a number of subcell hierarchies in the database.

In the manufacture of a chip, the various materials which make up the electrical components of the chip are applied in layers with masks which specify exactly where each material will be placed. Many circuit design systems present these masks to the designer exactly as they will appear on the chip. This gives the designer little information about the actual electrical function of the design. Magic, however, abstracts these mask layers into a style referred to as *logs*. These logs are similar to a symbolic circuit layout which a designer uses to understand the electrical functionality of the circuit. The primary difference is that each component is seen in its actual size and location. (15)

2.6 Summary

Based on the characteristics of engineering design systems and the key components of the object-oriented computing paradigm, object-oriented database management systems appear well suited for supporting these design systems. Some attempts have been made to demonstrate the improved performance of OODBMSs, but none of these attempts has successfully modeled a typical design application to measure this performance. The Magic layout design system uses a circuit design representation which is well suited for an object-

oriented database system. The ObjectStore database management system is one of the few commercially available OODBMSs available. It supports most of the utilities expected of an object-oriented database system and is suitable for implementation of the Magic layout system. Using Magic and ObjectStore, this thesis provides a typical design application for measuring object-oriented database performance.

III. Design and Implementation of Magic Using ObjectStore

3.1 Overview

The implementation of Magic with ObjectStore requires application of recognized software engineering principles. The first step is to determine the design of the Magic system and which components require modification. This design is then modified so that it can be implemented with ObjectStore. For performance measurement, the code is instrumented with timing commands where appropriate. Testing is then accomplished to verify functionality and to compare performance with the original Magic system.

3.2 Magic Design Recovery

Magic is a large software system consisting of over 250,000 lines of *C* source code in over 40 separate Unix directories. To maximize code efficiency, many of the data structures and algorithms are extremely complex, often using obscure *C* language characteristics. The design documentation consists of a maintainer's manual with a brief description of the directory layout and functionality along with in-line comments in the source code. As such, understanding the system organization and design is a difficult undertaking.

In approaching a design recovery of the Magic software system, the first step is to review the existing documentation. This review reveals a combination of functional and object oriented problem decomposition. The object-oriented modules, such as the *window manager* and *database manager*, encompass all data structures and services for an object completely within the module. Other modules like *plot*, *plow*, and *wiring* include all procedures necessary for accomplishing the designated function.

Each module has its own subdirectory except for some utility and other miscellaneous functions which are grouped together into combined subdirectories. To simplify understanding of the overall Magic system, these modules have been grouped into "super-modules" which represent the main services provided by Magic. Of greatest importance to this thesis is the database management super-module. Most interfaces to the ObjectStore database management system will take place within this module. Figure 2 shows each of these super modules and indicates whether a module interfaces with the database

management module. In this figure, the label at the top of the box represents the super-module name and the lower portion of the box lists the individual modules within that super-module. Lines to each individual module indicate interaction with the database manager.

While most modules interface with the database manager, implementation with ObjectStore affects only a few of these interfaces. Most of the ObjectStore administration (such as opening and closing the database) must occur within the *main* module. Since the window manager does all window manipulation associated with displaying a circuit that has been retrieved from the database, its performance is closely linked with the database manager. To determine the effect a command will have on the database, and to ensure this command continues to have the same effect with ObjectStore, understanding of the *command interpreter* is also important. Finally, the *utilities* module is of concern since it includes functions for abstract data types such as hash tables and lists.

Analysis of the *C* header files for the database manager reveals the primary data structure shown in Figure 3. Here an object is represented by a box with the object name in the top section of the box and its attributes in the lower section. Diamonds represent relationships between objects. The key component of a Magic circuit is the *cell definition* (`CellDef`). This includes descriptive fields such as the cell name, associated chip technology, and time of last update. It also includes pointers to the *planes* which contain the geometrical representations in the cell; a pointer to a list of *labels* associated with the cell; a pointer to a list of cell instances (referred to in Magic as *cell uses*); and a pointer to a hash table of all instances of other cells which reference the cell. Each plane also has pointers to a corner-stitched list of *tiles* which are contained in the plane. Normally `ti_body` specifies the paint in the tile; however, tiles in the subcell plane include a list of pointers to the subcell uses which overlap the tile (`CellTileBody`). All cell definitions currently active in Magic are contained in `dbCellDefTable`, a hash table which is visible only to the database manager.

To better understand the relationship between cell definitions and cell uses, Figure 4 provides a simplified example. Cells `A` and `B` each represent cells that have cells `X` and `Y` as subcells. The cell definition of `X` is named `CdX`. It references cell use `CuX3` with its

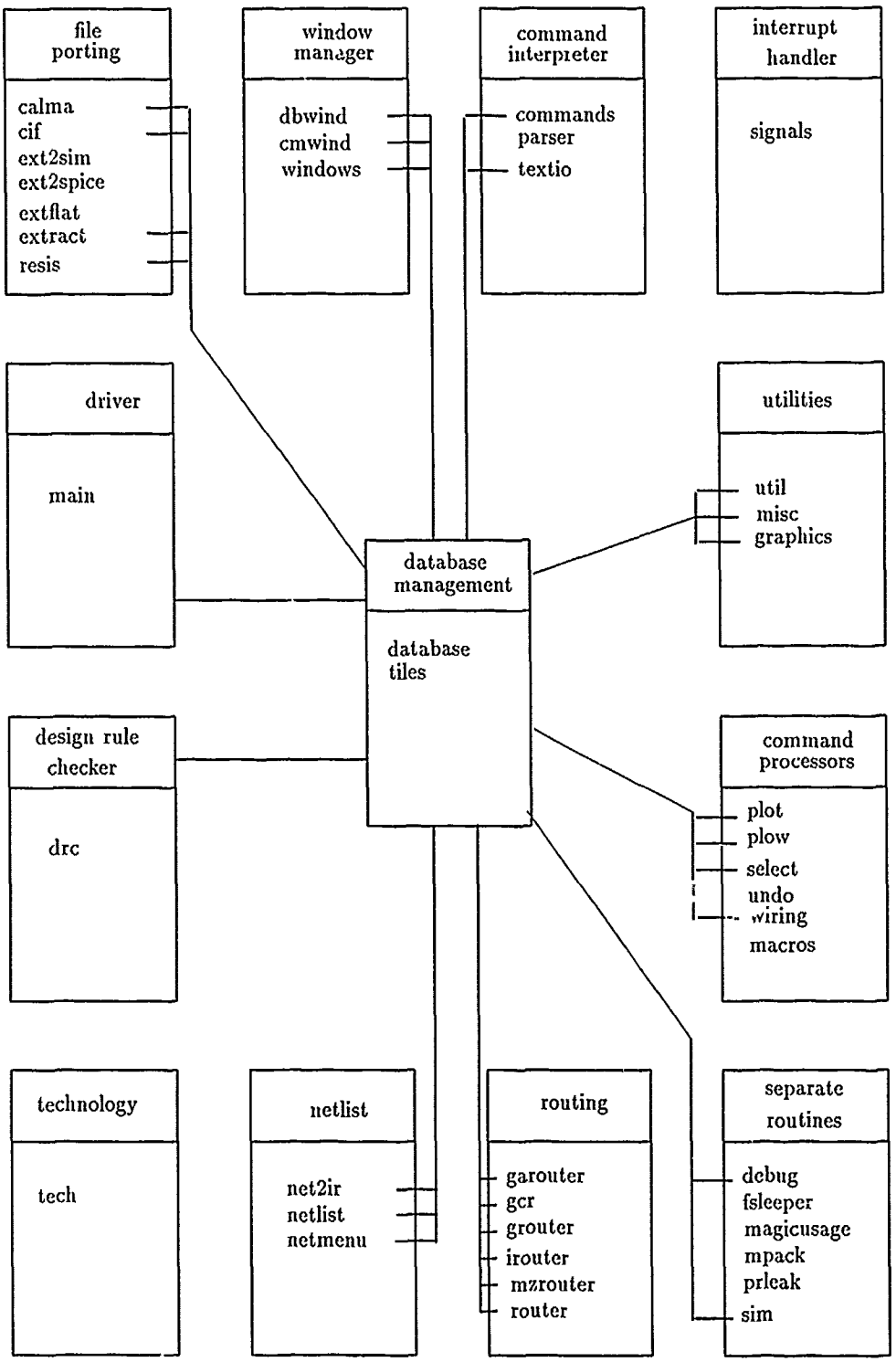

Figure 2. High-level Magic directory organization

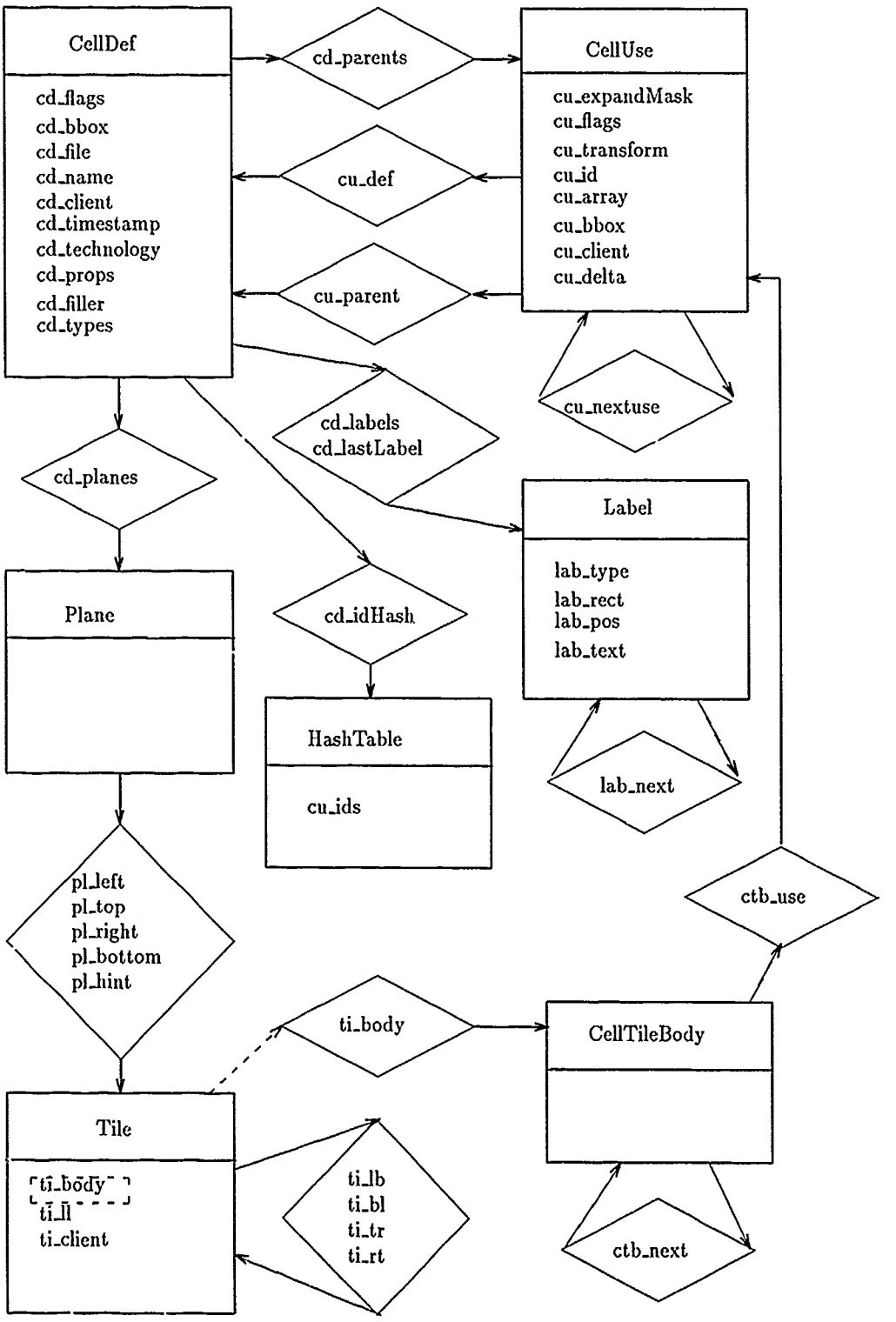

Figure 3. Data structure for Magic cells

cd_parents pointer. It also points to a hash table which only includes cell use CuX1. This indicates that CdX is not referenced by any cells other than itself. Cell use CuX3 is the cell use associated with cell B. The cu_parent pointer to cell definition CdB shows this relationship. The cu_def pointer in CuX3 points to cell definition CdX, of which CuX3 is a specific instance. Each cell use also has a cu_nextuse pointer which points to the next cell use associated with a particular cell definition. In this case, CuX3 points to CuX2 which points to CuX1 where the list terminates with a null pointer. CuX2 represents the instance of CdX in cell A and CuX1 represents the instance of CdX associated with cell X. Each cell definition always has an instance associated with itself. Cell definition CdA provides a better example of the cd_idHash pointer. In CdA this points to a hash table of all cell uses that are included in cell A. This includes an instance of the cell itself (CuA1) as well as each of its subcells (CuX1 and CuY1).

While the basic functionality of the database module can be determined from the limited documentation in existence and its data structures can be determined from the header files, actual understanding of the call hierarchy and effect of commands can only be accomplished by tracing the path of input commands through the Magic system. Some additional information is also obtained through frequent use of the Unix *grep* and *calls* commands. The functions of the database module most likely to be affected by implementation with ObjectStore are described below. The source code files associated with these functions are listed in parentheses.

- Create and delete cell definitions and cell uses (DBcellname.c).
- Create and maintain the cell definition table (DBcellname.c).
- Write and read cells to and from disk (DBio.c).
- Create, split, join, and delete cell planes and tiles (tile.c).

If the database module was truly object-oriented, design recovery would end here. Unfortunately, many modules throughout Magic directly access the data structures of the database module, thereby making the interface less well defined. For instance, the initialization routines in many modules directly access planes within a cell. Similarly, each

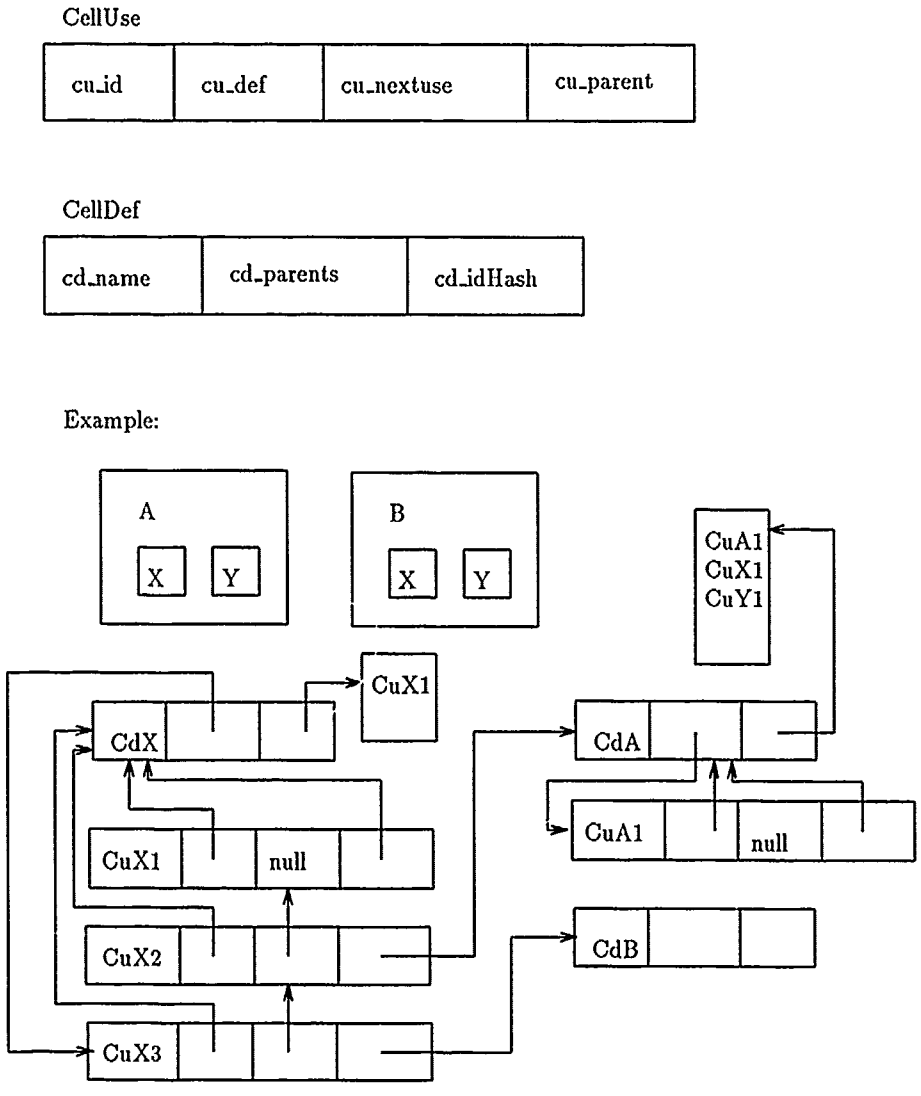

Figure 4. Relationship between cell definitions and cell uses

window has a cell instance associated with it which the window manager modifies without going through the database module.

To test and debug the database module, one must understand the data flows between the window manager, command processors, and database manager. The database window manager command interpreter (`DBWcommands` function in program `DBWprocs.c`) is the key procedure in this process. If a button is pressed, the current button handler is activated from within the database window manager to interpret the button, perform the desired processing, and update the window as appropriate. If a text command is entered, it is first parsed by the text processor (*textio* module). The database window manager then activates the selected command processor to accomplish the requested processing. Any updates to the window are then initiated from the command processor.

3.3 Restructuring of Magic Software to Work with ObjectStore

For this thesis, the only modifications made to the Magic software are those necessary for Magic to work with the ObjectStore database management system. This requires persistently allocating all transient database structures and providing an entry point into the ObjectStore database. In addition, the ObjectStore database file must be opened and closed and transactions must be specified such that all database accesses occur within a transaction. This first phase of implementing Magic with ObjectStore requires the following changes. (A complete summary of all modifications to the Magic software is contained in the *OSmagic Programmers' Manual* (5)).

- Where a database element such as those in Figure 3 has been allocated with the procedure `MALLOC`, remove the `MALLOC` and replace it with the ObjectStore Data Definition and Manipulation Language (DML) persistent `new` command.

```
/* replace MALLOC with ObjectStore DML new
 *
 * MALLOC(CellDef *, cellDef, sizeof (CellDef));
 */

cellDef = new(magicdb1, cellConfig) CellDef;
```

Similarly, where a database element is deallocated with FREE, remove the FREE command and replace it with a DML persistent delete command.

```
/* replace old memory deallocation with DML delete
 *
 * FREE (cellUse->cu_id);
 */

delete cellUse->cu_id;
```

- Make the cell definition symbol table (dbCellDefTable) the entry point into the database. This is accomplished by declaring it as a persistent variable.

```
persistent<magicdb1> osHashTable *dbCellDefTable = NULL;
```

- Declare and initialize (open) the database in the program main.c and include the main procedure within an ObjectStore transaction. The original attempt at accomplishing this was to enclose the contents of *magicMain* (the main Magic procedure) within a DML do_transaction statement; however the main Magic procedure is terminated from within another module, thus preventing the entire main procedure from executing. As a result, the do_transaction statement never ends and no data is written to the database. Resolution requires separate transactions to initialize Magic and another transaction (main_tx) for the main Magic process. Note that a different format is used for the main transaction. ObjectStore allows a transaction to be enclosed within the do_transaction statement or started with transaction::begin and ended with transaction::commit (to save the results of the transaction) or transaction::abort (to restore the database to its state prior to the transaction). Code modifications to open the database and implement ObjectStore transactions are shown in Figure 5.

- Close the database and commit the main transaction in the procedure *MainExit.*

```
transaction::commit(main_tx);
magicdb1->close();
```

In an object-oriented *C++* program, implementation of these changes may have been rather straightforward and uncomplicated. Unfortunately, Magic is not such a program.

```
/* Open database "/osmagic/magicdb1" */
magicdb1 = database::open("/osmagic/magicdb1",0,0664);

do_transaction() {
    workspace::set_current(user_ws);
}  /* end of transaction */

  /* begin initialization transaction */
do_transaction() {

    mainInitBeforeArgs(argc, argv);
    mainDoArgs(argc, argv);
    mainInitAfterArgs();
}  /* end of initialization transaction */

  /* begin main transaction called "main_tx" */
main_tx = transaction::begin(transaction::update);


TxDispatch( (FILE *) NULL);

mainFinished();
```

Figure 5. Code modifications to function *magicMain*

The first difficulty is that Magic is written in *C* rather than *C++*. *C++* was designed to be compatible with *C* (18); however, this compatibility is not complete. ObjectStore has a *C* library interface, but this does not allow one to take full advantage of object-oriented programming techniques. Any program which contains ObjectStore DML must be compiled with the DML compiler. This compiler is a slightly modified *C++* compiler. Thus all procedures in a program containing ObjectStore DML, whether affected by ObjectStore or not, must be modified for compatibility with *C++*. This requires type specification of all function parameters. In many cases, Magic passes function pointers as parameters, which complicates type specification. Also, for a *C++* procedure to be linked with a *C* program, the linkage must be specifically defined with an `extern "C"` type specification for the function declaration. Thus, all of the forty plus header files containing *C++* function declarations must be modified to include the `extern "C"` qualifier. This is accomplished by defining a preprocessor constant of `_OSMAGIC` in a header file (`osmagic.h` for the programs using ObjectStore DML). Other header files are then modified to include the `extern "C"`

```
#ifndef _OSMAGIC
/* if using old magic code, use "C" function declarations */
/* Jacobs 02/08/91 */

extern Plane *TiNewPlane();
extern void TiFreePlane();
extern Tile *TiSplitX();
extern Tile *TiSplitY();

#else
/* use function declarations modified for compatibility with C++ */
/* Jacobs 02/08/91 */

extern "C" Plane *TiNewPlane(Tile*, CellConfig*);
extern "C" void TiFreePlane(Plane*);
extern "C" Tile *TiSplitX(Tile*, int);
extern "C" Tile *TiSplitY(Tile*, int);

#endif
```

Figure 6. Sample header file modifications from **tile.h**

qualifier for DML programs if _OSMAGIC is defined or just the qualifier extern if _OSMAGIC is not defined. Figure 6 shows modified code from `tile.h` which demonstrates the changes required for each header file.

Other procedures outside of the database module also must be modified due to deficiencies in Magic's object-oriented implementation. Additional storage is allocated for a cell in the string duplication (`strdup.c`) program of the utilities module. A persistent version of this program (`osstrdup.cc`) is required for any string duplication within the cell structure. The cell labeling (`DBlabel.c`), cell painting (`DBpaint.c, DBtiles.c`), and cell subroutine (`DBcellsubr.c`) programs also allocate cell storage which must be persistent.

Since the cell symbol table is used as a database entry point, it must also be persistently allocated. The hash table abstract data type (ADT) (`hash.c`), of which the cell symbol table is an instance, is also used for non-database functions, so a separate, persistent hash table ADT (`oshash.cc`) must be created and used for the cell symbol table. This symbol table only uses strings as keys, so the *C* union in the original hash table ADT can be replaced with a string. This simplifies DML implementation by eliminating the need for union discriminant functions.

The maze router (`mzrouter`) initialization procedure attempts to directly access planes within a cell. The first time the initialization procedure is run on the database, the cell and planes become persistently allocated. The `mzrouter` initialization procedure must then be modified to access the persistent planes by traversing the cell hierarchy.

Modifications up to this point have been necessary for Magic to work with persistently allocated structures in an ObjectStore database. Previously these structures were transiently allocated and could be cleared by quitting Magic or by reading the cell again. Persistent allocation eliminates the possibility of deleting a cell or removing changes that are unwanted. To provide these functions and other input/output functions in a manner similar to the original Magic system, ObjectStore's versioning capabilities are required.

Versioning requires definition of a `configuration` to be versioned and persistent allocation of `workspaces` to control the versions. For this implementation of Magic, the global workspace contains the latest frozen version of a cell, much as the flat disk file does in the original version of Magic. New cells are created and modified in the user workspace. The user workspace is set as the current workspace for the duration of the Magic program.

The obvious choice for a version configuration is a cell definition and all of its subcomponents and subcells. A symbol table must be created for the cell configurations so that each cell definition can be associated with its configuration. Whenever a cell is read, it is checked out of the global workspace into the current user workspace. Writing a cell requires checking the cell back into the global workspace. To flush a cell, the version in the current user workspace is destroyed and the old version is checked out of the global workspace to replace the destroyed version in the user workspace. New procedures which search the configuration symbol table and check the appropriate configuration in or out of the current workspace are needed.

Even with versioning, some commands can not be made to work exactly like the original Magic system. Both the *write* and *save* command write a cell to disk. The *save* command allows the cell to be saved under another name. Both of these commands allow further modification of the cell after writing. These commands must be modified to work with ObjectStore. *Save* will check the cell back into the global workspace, but will allow

continued editing of the cell by checking it back out. *Write* will check the cell into the global workspace and prohibit further editing.

In the original Magic, cells are deleted by using the Unix *rm* command external to Magic. This is not possible using ObjectStore so the new command *remove cellname* must be added. This command will only delete a cell definition if it is not used by any other cells.

3.4 Code Instrumentation for Performance Measurement

To compare the performance of Magic with and without ObjectStore, a method must be provided for measuring this performance. The *Sun* operating system provides profiling options which are implemented by the compiler. This profiling provides detailed timing and usage statistics on every function in Magic. Unfortunately, due to the size of Magic, interpretation of this information is time intensive and complicated. Since such detailed information is not necessary, the code itself is instrumented at critical points to measure only that information which is essential for comparing performance of the two versions of Magic.

All commands and button input are processed through the command interpreter module. The `TxDispatch` function in the program `txCommands.c` dispatches all Magic text and button commands. This allows timing statistics to be initiated prior to calling the command processor or button handler and terminated immediately following the call. Similarly, initialization statistics are measured by surrounding Magic's initialization procedures with statistical commands. A program (`CommandStats.c`) is written for gathering statistics. `CommandStats` makes calls to the Unix functions *getrusage* and *gettimeofday* and calculates processing time and wall clock time. This information is printed to a file along with the command being processed or the button handler in use.

3.5 Testing

3.5.1 Testing Magic Functionality. If ObjectStore is to replace the existing database management system of Magic, it must maintain complete functionality of the original

system. To exhaustively test the functionality of a system the size of Magic would require an inordinate amount of time. Fortunately, since the purpose of this thesis is only to demonstrate the feasibility of ObjectStore, such exhaustive testing is not necessary. Instead, a simple, sound test of the database functions along with limited testing of the other Magic functions should be adequate.

Part of the limited documentation of Magic is a tutorial which walks the new user through all of the basic commands that are available. This tutorial is the basis of the functionality testing for Magic. Where database commands are used such as *save*, *flush*, and *load*, an attempt is made to test all classes of input parameters (e.g. *save* is tested with no parameters, with the same cell name as a parameter, and with a new cell name as a parameter). Since few new functions have been added and most of the Magic modifications are simply conversions from *C* to ObjectStore DML, little should change in Magic's functionality. Any errors introduced as a result of the changes should be significant enough to be caught by the above tests. In addition, since performance testing concentrates on the database (see the following section), these tests serve to further validate the functionality.

3.5.2 Performance Testing. Performance testing compares the differences in access time between Magic implementation using ObjectStore and the original implementation using flat Unix files. This testing must measure Magic performance during a typical user session along with concentrated testing of the database functions of Magic. A typical user session (as used at the Air Force Institute of Technology) does not require many database accesses. This is likely due to the difficulty of accomplishing common database functions (such as searching for an existing cell) and the experimental nature of educational research which leads to circuits built entirely from scratch. Since a typical user session does not adequately test the database capabilities, performance comparisons are also conducted using the HyperModel Benchmark (see Section 2.3.1) guidelines. The HyperModel Benchmark lists six areas which are important for measuring database performance - lookup and retrieval, traversal, insertion, sequential scan, closure operations, and opening and closing of the database.

All performance testing is accomplished with existing Magic commands. As such, some of the six areas above may be only partially tested. Use of existing commands is necessary so ObjectStore performance can be directly compared to the existing flat file structure. The ObjectStore database is loaded in advance with all Magic cells in a specific search path, so that the search space is roughly equivalent with that of the Unix directories.

Performance testing is accomplished on two existing cells. One cell contains 87 subcells with eight levels of nesting and the other cell contains two subcells with one level of nesting. The various Magic commands for measuring the benchmark criteria are discussed below.

- Look up and retrieve an object from the database. The *load cellname* command searches the database until it finds the specified cell and then displays it in the selected window. If the cell has any subcells, these are not initially displayed.
- Traverse pointer hierarchy. The *expand* command loads and displays all of the subcells in a selected area of the root cell. When the entire cell is selected, all of the subcell pointers are dereferenced and the subcells displayed.
- Insert an item into the database. The *getcell cellname* command loads a subcell from the database and creates a new instance of that cell in the root cell. The *load* command without a specified cell creates a new cell definition in the database. With the non-persistent version of the database, the new cell definition or instance is not actually created until the cell is saved; therefore, the time to write modified cells to disk must be included in the comparison with the ObjectStore version.
- Closure operations. In normal operation, the Magic design rule checker runs in the background. To test closure, however, the design rule checker is turned off, a subcell is added on top of the existing cell, and the design rule checker is run on the entire cell.
- Sequential Scan. Since no use is made of ObjectStore collections, this aspect of the database benchmark is not tested.

- Database initialization. Since the existing Magic system has no database, this benchmark can not be directly compared; however, the tests are still run and compared based on the overall time to initialize Magic.

3.6 Summary

The size and complexity of Magic presents a difficult system to understand and modify. Since ObjectStore is written to take advantage of the object-oriented characteristics of *C++*, Magic's imperfect object-oriented programming structures and its use of obscure *C* programming routines increase the difficulty of converting it to ObjectStore. Since the original Magic has no database management system whatsoever, performance testing is complicated by the inability, in many cases, to directly compare the ObjectStore version of Magic with the original Magic version. Many of these difficulties are overcome with careful implementation of ObjectStore utilities and with selective testing using Magic commands in a well prepared test environment.

IV. Results Analysis

4.1 Overview

The primary objective of this thesis is to show that an object-oriented database can provide the performance and functionality necessary to support a computer-aided design tool. By careful application of ObjectStore utilities, complete functionality of the Magic VLSI circuit layout system has nearly been obtained. This chapter compares the relative performance of Magic as implemented with ObjectStore to the original flat file data management system. The chapter also points out the difficulties encountered while converting the complex *C* code of Magic to work with the complicated, object-oriented data management facilities of ObjectStore.

4.2 Performance Comparison of ObjectStore and Flat Data Files

The performance requirements of a complex engineering design system such as Magic are immense. To satisfy these requirements, a database management system must perform nearly as well or better than a flat file data management system. To compare the performance of Magic using ObjectStore to Magic's performance using its original flat file system, the performance tests described in Section 3.5.2 were accomplished using two different Magic databases.

drfm This database consists of 110 objects (i.e., cell definitions) and 1861 different instances (i.e., cell uses) of these objects. There are over 78,000 fundamental data items (i.e., tiles) at the lowest level of the object hierarchy. This database was tested using cell **drfmchip** with 87 subcells and eight levels of nesting. The Magic display of this cell, with all subcells completely expanded, appears in Figure 7.

tutorial This database contains 70 objects and 103 different instances. There are nearly 9300 fundamental data items at the lowest level of the object hierarchy. Testing was accomplished using cell **tut4a** which contains two subcells and one level of nesting. Figure 8 shows the Magic display of the tiles and subcell structure of this cell.

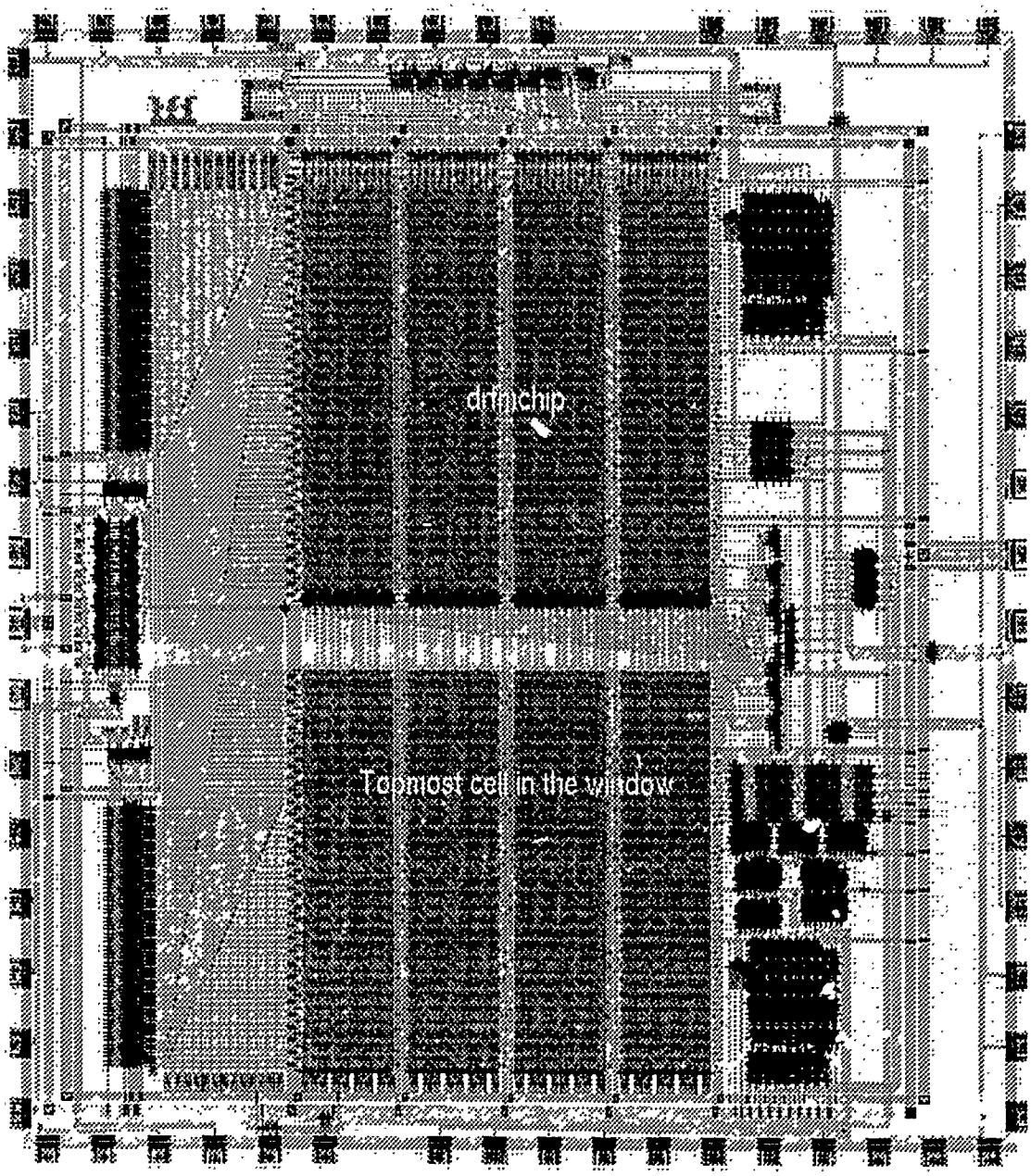

Figure 7. Magic display of cell **drfmchip**

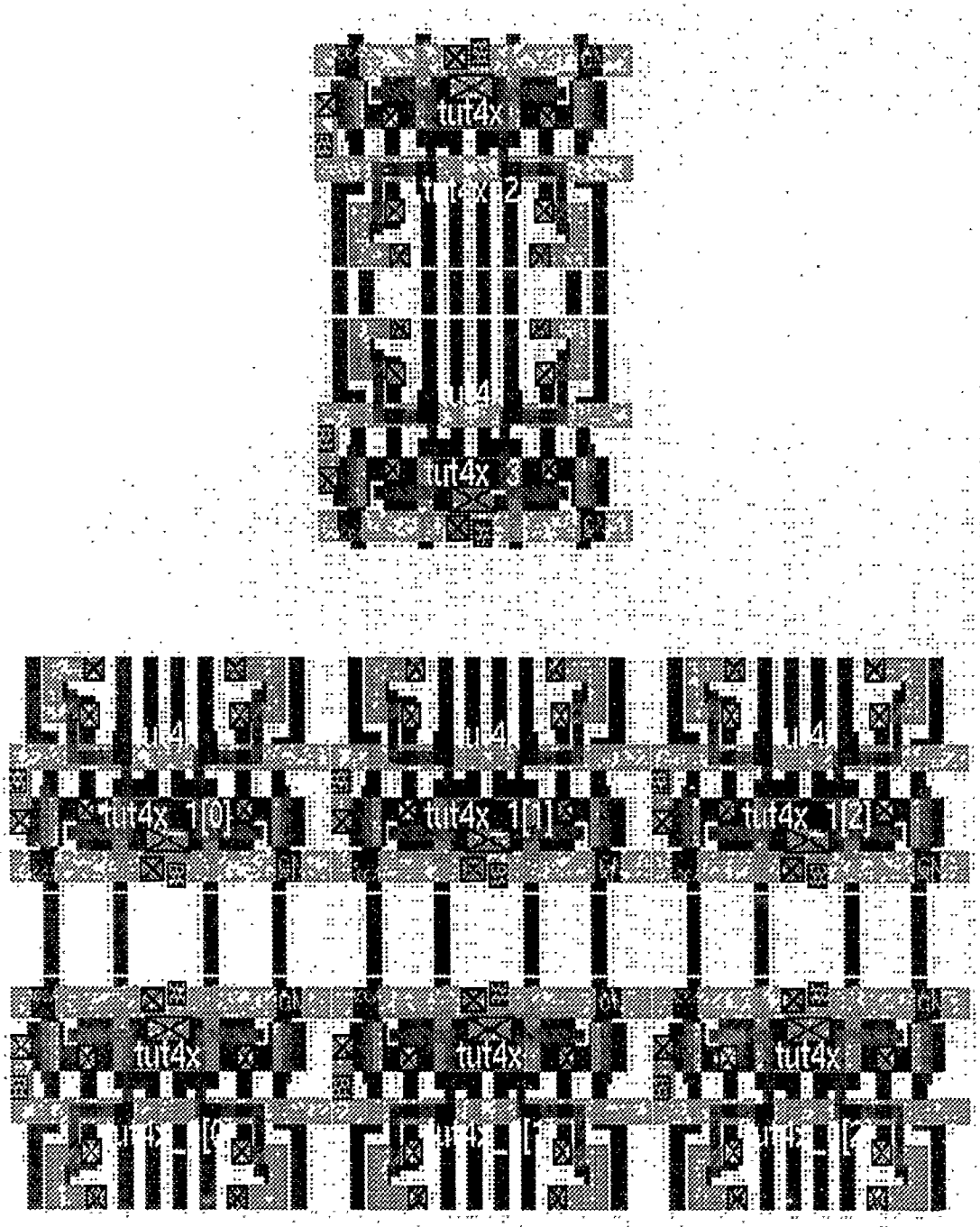

Figure 8. Magic display of cell tut4a

The results of the performance tests are shown in Tables 2 and 3. Raw performance test results are contained in Appendix A. The results shown in Tables 2 and 3 represent averages of all valid results obtained for each command. For more accurate comparison with ObjectStore, the resources necessary for writing all modified cells to disk are added to the performance results for the insertion and closure tests.

For the `drfmchip` cell, instance insertion and closure are tested on a subcell nested four levels deep in the hierarchy (`big_nandmux`) and on a subcell nested eight levels deep (`mcell10`). There are 16 instances of the `big_nandmux` subcell and 6144 instances of the `mcell10` subcell. Between these two levels of nesting, the time difference to accomplish instance insertion and closure was considerable (See Table 4). The average of these two different levels is used for comparison in Table 2. Instance insertion and closure are tested on a subcell nested only one level deep in the `tut4a` cell. There are only eight instances of this subcell.

Testing of the **drfm** database was accomplished with ObjectStore cache sizes of both 640 and 2048 sectors. The 640 sector cache worked more quickly for look up and retrieval, traversal, initialization, and object insertion. These are the results shown in Table 2. Similarly, the results for the 2048 sector cache are used for instance insertion and closure. All ObjectStore results for the **tutorial** database were obtained with a cache size of 2048 sectors.

The results for the different cache sizes are summarized in Table 5. Instance and closure results in this table are based on the subcell `big_nandmux` which is nested four levels deep. Neither size of cache performs better for all test cases. Optimal cache size will depend on which database utilities are more commonly used.

The commands used for performance testing fall into three different categories. Those used for *look up and retrieval* and *hierarchy traversal* require reading from the database. In the original version of Magic, *Insert* and *closure* commands are performed entirely within memory. For this reason, the resources necessary to write to disk all cells modified by these commands are added to the results in Tables 2 and 3. *Initialization* is a combination of reading and writing from the database and initializing memory. In general, Object-

Criteria Tested *Command Used*	Resource Measured	Data Management System		Percent Change
		Flat File	ObjectStore	
Look up/retrieve *load drfmchip*	CPU time (seconds)	0.06	0.05	-17
	Elapsed time (seconds)	0.12	0.33	+175
	Page Faults with I/O	2	0	-∞
	Page Faults without I/O	4	36	+800
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	6.62	1.54	-76
	Elapsed time (seconds)	9.32	4.37	-53
	Page Faults with I/O	89	0	-∞
	Page Faults without I/O	733	380	-48
	Disk Blocks In	91	0	-∞
	Disk Blocks Out	3	0	-∞
Insert (object) *load test*	CPU time (seconds)	0.03	0.02	-50
	Elapsed time (seconds)	0.124	0.040	-68
	Page Faults with I/O	0	0	0
	Page Faults without I/O	13	23	+77
	Disk Blocks In	0	0	0
	Disk Blocks Out	6	0	-∞
Insert (instance) *getcell test*	CPU time (seconds)	1.36	0.22	-84
	Elapsed time (seconds)	2.63	0.24	-91
	Page Faults with I/O	1	0	-∞
	Page Faults without I/O	112	15	-87
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	70	0	-∞
Closure *drc catchup*	CPU time (seconds)	52.19	193.42	+271
	Elapsed time (seconds)	53.98	197.66	+266
	Page Faults with I/O	7	2	-71
	Page Faults without I/O	136	151	+11
	Disk Blocks In	1	0	-∞
	Disk Blocks Out	70	0	-∞
Initialization	CPU time (seconds)	5.74	5.86	+2
	Elapsed time (seconds)	10.71	15.21	+42
	Page Faults with I/O	82	28	-66
	Page Faults without I/O	203	493	+143
	Disk Blocks In	13	8	-38
	Disk Blocks Out	1	2	+100

Table 2. Benchmark performance results for **drfm** database

Criteria Tested *Command Used*	Resource Measured	Data Management System		Percent Change
		Flat File	ObjectStore	
Look up and retrieve *load tut4a*	CPU time (seconds)	0.01	0.03	+200
	Elapsed time (seconds)	0.047	0.037	-21
	Page Faults with I/O	1	0	-∞
	Page Faults without I/O	6	30	+400
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	0.04	0.02	-50
	Elapsed time (seconds)	0.0788	0.0252	-68
	Page Faults with I/O	1	0	-∞
	Page Faults without I/O	3	6	+100
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Insert (object) *load test*	CPU time (seconds)	0.020	0.017	-15
	Elapsed time (seconds)	0.158	0.026	-84
	Page Faults with I/O	0	0	0
	Page Faults without I/O	13	16	+23
	Disk Blocks In	0	0	0
	Disk Blocks Out	7	0	-∞
Insert (instance) *getcell test*	CPU time (seconds)	0.060	0.013	-78
	Elapsed time (seconds)	0.532	0.029	-95
	Page Faults with I/O	0	0	0
	Page Faults without I/O	14	1	-93
	Disk Blocks In	0	0	0
	Disk Blocks Out	22	0	-∞
Closure *drc catchup*	CPU time (seconds)	0.58	2.00	+245
	Elapsed time (seconds)	1.05	2.03	+93
	Page Faults with I/O	0	0	0
	Page Faults without I/O	24	6	-75
	Disk Blocks In	0	0	0
	Disk Blocks Out	22	0	0
Initialization	CPU time (seconds)	5.66	5.82	+3
	Elapsed time (seconds)	8.53	9.91	+16
	Page Faults with I/O	14	4	-71
	Page Faults without I/O	299	514	+72
	Disk Blocks In	2	4	+100
	Disk Blocks Out	1	1	0

Table 3. Benchmark performance results for **tutorial** database

Criteria Tested *Command Used*	Resource Measured	Level of Nesting		Percent Change
		8 (`mcell10`)	4 (`big_nandmux`)	
Insert (instance) *getcell test*	CPU time (seconds)	0.41	0.022	-95
	Elapsed time (seconds)	0.45	0.027	-94
	Page Faults with I/O	1	0	$-\infty$
	Page Faults without I/O	29	1	-97
Closure *drc catchup*	CPU time (seconds)	368.61	18.23	-95
	Elapsed time (seconds)	376.75	18.56	-95
	Page Faults with I/O	3	2	-33
	Page Faults without I/O	251	51	-80

Table 4. ObjectStore performance at two different levels of subcell nesting

Store had better response time than Magic's original data management system for those commands with frequent database access relative to total processing time.

ObjectStore's performance varies most significantly with the *closure* command (i.e., *drc catchup*). To understand why, more insight into the design rule checker (*drc*) is required. The design review checker applies rules from a file of over 500 lines to each tile in the cell. Hierarchical designs are checked by ensuring the cell alone is consistent, and that the combination of the cell and all of its subcells is consistent (14). This may require traversing the cell hierarchy a number of times to complete the design rule checking; thus, small discrepancies in response time are multiplied into large differences such as those shown in Tables 2 and 3.

The design rule checker usually runs in the background because of the large amount of processing it requires (14). It limits its checks to those cells which are currently in memory; other cells are checked the next time they are read into memory. This allows incremental application of design rules to the cell and eliminates the need to process the entire cell at once. Since the ObjectStore version of Magic extends memory to include database items which are on the disk, it always appears as if the entire database is in memory; thus, the design rule checker checks all cells in the database. Because of these variations in virtual memory, valid comparisons can only be made with the entire cell resident in main memory.

The ObjectStore database required considerably more disk space than Unix flat files (see Table 6). Both ObjectStore and Magic add overhead to the ObjectStore database

Criteria Tested *Command Used*	Resource Measured	Cache Size (sectors)		Percent Change
		640	2048	
Look up/retrieve *load drfmchip*	CPU time (seconds)	0.05	0.07	+40
	Elapsed time (seconds)	0.33	0.35	+6
	Page Faults with I/O	0	0	0
	Page Faults without I/O	36	43	+19
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Hierarchy Traversal *expand*	CPU time (seconds)	1.54	1.61	+4
	Elapsed time (seconds)	4.37	5.91	+35
	Page Faults with I/O	0	0	0
	Page Faults without I/O	380	467	+23
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Insert (object) *load test*	CPU time (seconds)	0.022	0.021	-4
	Elapsed time (seconds)	0.040	0.053	+32
	Page Faults with I/O	0	0	0
	Page Faults without I/O	23	20	-13
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Insert (instance) *getcell test*	CPU time (seconds)	0.05	0.02	-60
	Elapsed time (seconds)	0.077	0.027	-65
	Page Faults with I/O	1	0	$-\infty$
	Page Faults without I/O	10	1	-90
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Closure *drc catchup*	CPU time (seconds)	23.48	18.23	-22
	Elapsed time (seconds)	23.87	18.56	-22
	Page Faults with I/O	2	2	0
	Page Faults without I/O	20	51	+155
	Disk Blocks In	0	0	0
	Disk Blocks Out	0	0	0
Initialization	CPU time (seconds)	5.86	6.02	+3
	Elapsed time (seconds)	15.21	15.47	+2
	Page Faults with I/O	28	54	+100
	Page Faults without I/O	493	481	-2
	Disk Blocks In	8	13	+62
	Disk Blocks Out	1	2	+100
	Disk Usage (kbytes)	724	4882	+574

Table 5. ObjectStore performance comparison with two different cache sizes

Data Item	Size in Kbytes		Percent Change
	Flat File	ObjectStore	
`tutorial` database	58	713	+1129
`drfmchip` database	724	4882	+574
initialized database (empty)	0	147	$+\infty$
`test` object & instance	0.081	2.3	+2740

Table 6. Comparison of ObjectStore and Unix flat file disk usage

which is not saved in the Unix file representation. For Magic this overhead is approximately 50 kilobytes. The ObjectStore overhead varies with the total size of the database. With no data other than the Magic and ObjectStore overhead in the database, the total database size is nearly 150 kilobytes. In addition, ObjectStore holds the entire data structure for each cell in memory. This includes pointers and empty hash table buckets. The Unix files only contain the contents of the cell data structure. Because of the large amount of overhead associated with ObjectStore, the difference is disk usage is more significant for smaller data items.

4.3 Conversion to ObjectStore

One of the objectives of this thesis was to show that conversion of a design system to an object-oriented database is a cost-effective venture. If this new technology is to replace existing design systems, system managers must be convinced that the benefits of using a database will outweigh the costs of conversion. While performance is one of the key issues in this decision, there are a number of benefits (as discussed in Chapter 1) and costs associated with converting to a database management system.

4.3.1 Problems Encountered. Due to the size and complexity of Magic, and the intricacies of ObjectStore, a number of difficulties were encountered during the conversion process. These difficulties were further aggravated by the immaturity of the ObjectStore database system and the lack of documentation for Magic.

Implementation of ObjectStore's versioning capability presented the only problem which could not be resolved. The `configuration::of(`*object*`)` command is designed to

return a pointer to the configuration of the object specified. In a number of places this pointer is then used to allocate a subobject in the same configuration as the object. The sequence of command is as follows:

```
cellConfig = configuration::of(cellDef);
cellUse = new(magicdb1, cellConfig) CellUse;
```

ObjectStore fails when attempting to allocate the subobject. No fix is currently available for this problem, so versioning does not work with the current ObjectStore implementation of Magic. This prevents a circuit designer from backing out of cell modifications — once a cell is changed, the change is reflected permanently in the ObjectStore database.

Another problem area stems from the incompatibility of *C* and *C++*. In theory, *C++* is designed to be a superset of *C* (18). While this may be true in a *C* program that is very well designed and coded, in reality *C* allows many structures that are not compatible with *C++*. While Magic is a rather well designed system, its size and the complex programming structures which it uses to maximize efficiency have led to code which is unstructured and difficult to trace. Some common problems include functions used without being previously declared, data types used in header files that are declared in a separately included header file, and functions with different types of parameters being passed as parameters to another function. Since *C++* has much stricter type checking, these problems had to be resolved if the program included any ObjectStore DML commands which required the *C++* compiler.

While not directly related to compatibility between *C* and *C++*, the ability to cast types in *C* presents significant problems. Magic uses this capability extensively. In one particular instance, `ti_body` is declared as a `char`; however, to handle subcells within a plane, a list of pointers to `CellUse` is cast to this variable. Initially, Magic sets the `ti_body` field to a `char`. If ObjectStore encounters a `char` value when it is expecting a persistent pointer, it is unable to dereference the pointer and the program aborts. Such errors are very difficult to trace due to the use of type casting.

Many difficulties were encountered due to peculiarities of the ObjectStore DML. Those which were hardest to resolve are listed below.

- Inconsistency of schemas. If the database schema is modified, the old database is no longer valid for that application (16). A procedure must be made for converting the old data to match the new schema, or the data is lost.
- Incompatibility of persistent types with *C*. Persistent types have an extra level of indirection which must be accounted for if using them in a *C* program (16). ObjectStore also uses a persistent dereferencing type which is not compatible with the *C* compiler. This requires all *C* programs to access persistent data types through a *C++* interface.
- Databases and persistent database entry points must be declared at a global level. Since Magic is broken into a number of subdirectories, it is unclear at exactly what level these declarations should occur. This was resolved by declaring them at file level in any directory and including an `extern` declaration in a header file.
- When using versioning, the effects of `configuration::forget` are unclear. This command should remove the specified version from the current workspace (12); however, this does not happen when the versioned object has just been created and exists in no other workspace. The `configuration::destroy_version` command was used instead. To be able to destroy a version, however, the configuration must have been previously frozen in some workspace. This is accomplished by first checking any new configuration into the global workspace before checking it back out to make modifications.
- A discriminant function is required when using a `union` (12). No explanation is given on what this function should do and when it should be called. To eliminate the associated complexities, the `union` was removed as it was no longer necessary for the updated Magic program.
- When more data is read into memory than ObjectStore can manage, it attempts to evict a page from memory causing an exception which crashes Magic. To overcome this problem when testing `drfmchip`, the cache size had to be manually increased to 2048 sectors.

Most of the facilities discussed above are included in the ObjectStore documentation, but inadequate explanation of their purpose and implementation is given. The documentation often tells how to implement an ObjectStore function with no explanation as to why it is done that way. As a consequence, when something must be done different for a specific application, it is difficult to make the adaptation. An example is the Makefile structure for Magic. Magic has a separate Makefile for each subdirectory. This Makefile compiles each program in the subdirectory and links the object files into a composite object file for the entire subdirectory. Each subdirectory object file is then linked together for the Magic executable. The description of ObjectStore Makefiles assumes all object and source code is in one directory (11). The solution was simple — to only use the ObjectStore linker for the executable and not for each individual subdirectory — but many hours of analysis, trial and error, and consultation with the ObjectStore programmers were required before the problem was resolved.

Another significant difficulty with the documentation occurs when using the ObjectStore DML. Even though this is the primary language for ObjectStore, the *ObjectStore User Guide* deals primarily with the ObjectStore *C++* interface library. In many cases the DML replaces multiple library calls, thus simplifying the programming and subsequent maintenance. Only brief descriptions of these DML commands are given, however, with few examples and little correlation to the functions performed by the library interface.

4.3.2 Effort Required. This thesis implements a minimal database version of Magic. Changes were made to replace the original flat file structure with an ObjectStore database while maintaining the same functionality. No attempt was made to take full advantage of ObjectStore's other features. The time spent on this conversion is broken down in Table 7. The estimates in this chart are based on the perceived difficulty of each process and the total time available for the conversion.

As reflected in this table, the conversion to ObjectStore took longer than expected. Since *C* constructs are supposed to be compatible with *C++*, the only code conversions expected were those necessary for the ObjectStore DML. The many other conversions which were actually required had a significant impact on the time required to make Magic

	Estimated	Actual
Magic design recovery	10	11
ObjectStore Familiarity	10	11
C++ Conversion	0	13
ObjectStore Persistence	15	17
ObjectStore Versions	10	20
Total	45	72

Table 7. Man days spent converting Magic to work with ObjectStore

work with ObjectStore and its *C++* compiler. The other major deviation in time is that required for implementing ObjectStore versions. This is an extremely challenging ObjectStore utility and the documentation is somewhat limited. In some cases, such as `configuration::forget`, the versioning commands do not work exactly as the documentation describes them. In other instances, for example, when Magic creates an instance of a cell for a window or initializes a plane for a tool like the circuit router, it was difficult to determine what version the new plane or cell instance should be in.

4.4 Summary

Performance tests were accomplished for six benchmark areas using the Objectstore version and the original version of Magic. While problems were encountered during the implementation of Magic with ObjectStore, the implementation was complete enough to compare performance of the two data management systems. The problems encountered range from failures of ObjectStore procedures to difficulties in making *C* code compatible with the ObjectStore DML compiler. Some problems were further complicated by inadequacies in ObjectStore documentation. These difficulties are reflected in the effort which was required to convert Magic to use the ObjectStore database management system.

V. Conclusions and Recommendations

5.1 Overview

This chapter summarizes the activities necessary for implementing and testing Magic with the ObjectStore database management system. The results presented in Chapter 4 are analyzed and conclusions reached regarding how well the objectives of Chapter 1 have been satisfied. Finally, recommendations for further research are presented which may help answer some of the questions raised by this thesis.

5.2 Summary of Research

This thesis directly converted the Magic layout design system to take advantage of the database facilities of ObjectStore. A design recovery of Magic revealed the database manager module to be the critical component of this conversion. The data structure used by the database manager consists of a cell definition for each VLSI circuit and a list of cell uses for each particular instance of the cell definition.

Conversion of Magic required the cell definition and cell use data structures to be persistently allocated using the persistent new command of ObjectStore. File input and output was eliminated since the data structures are no longer destroyed when Magic is shutdown. Some Magic commands were modified slightly to account for invalid data remaining in the database along with data which the designer intends to keep.

To compare the performance of the original Magic to the version modified to work with ObjectStore, timing routines were instrumented in the code to measure the time required to complete a Magic command. Routines were also added to measure disk access. Performance testing was accomplished using Magic commands which require extensive database access. Emphasis in performance testing was placed on those database attributes specified in the HyperModel Benchmark.

5.3 Conclusions

The objectives of this thesis require that the ObjectStore version of Magic provide the full functionality of the original version. Response time must not increase by more

than ten percent over the original version. This thesis also attempts to demonstrate that conversion of Magic from its flat file data management system to ObjectStore is a cost effective undertaking. The results of Chapter 4 are analyzed in the following subsections to determine whether these thesis objectives were met.

5.3.1 Database Functionality. As described in its documentation, ObjectStore provides support for complete functionality of Magic in a manner similar to the original version. While some commands may perform in a slightly different manner, the same functions are supported. Unfortunately, however, ObjectStore does not work entirely as described in its documentation. The primary difference is implementation of ObjectStore's versioning facilities. These are necessary to allow Magic to roll back to a previously baselined design. The versioning facilities do not work correctly with ObjectStore version 1.1, so the implementation of Magic for this thesis does not support design roll back.

5.3.2 Database Performance. The results presented in Chapter 4 show Magic to meet performance goals for only three of the six areas of performance benchmark testing: hierarchy traversal and object and instance insertion. Because of this, one may tend to conclude that ObjectStore's performance is inadequate for supporting complex engineering design systems such as Magic. R.G.G. Cattel suggests, however, that benchmark performance tests may not be an accurate measure of performance; rather "The most accurate measure of performance for engineering applications would be to run an actual application .... (2:364)"

When actually using Magic, the difference in performance was not readily apparent. For look up and retrieval, the difference in response time, while representing an increase of nearly 200 percent, was still only measured in fractions of a second — barely perceptible to a human user. Database initialization, while taking 42 percent more time with ObjectStore, is only performed once per user session. Five seconds in a session that may be hours long does not seriously detract from overall performance. Closure operations, as tested with the *drc catchup* command, took considerably longer with ObjectStore. Even with the original version of Magic, however, this command took a long time to accomplish. It is for this reason that the designers of Magic expect users to generally run the design rule checker

in the background. This background checking can be turned off when a large number of changes are made to a circuit design. When the changes are completed, *drc catchup* will run the design rule checker on all changes made during the session. Again, this represents a few minutes of trade-off in performance during what typically is an hours long session.

ObjectStore required a considerable amount of space to store the Magic databases. In the original environment which the performance tests were accomplished, this amount of space had a significant impact. With the current implementation of Magic using a single transaction, no segment of the database could be found to remove from memory once the entire circuit was swapped into memory. The single transaction implementation coupled with the large database size led to memory quickly being used up and the Magic system failing. These problems do not necessary reflect poorly on object-oriented databases, however, since proper transaction implementation would easily overcome the problems. In addition, newer releases of ObjectStore are projected to better handle such memory swapping problems (16).

When considering the performance of the ObjectStore version of Magic, one must realize that Magic was not designed using object-oriented programming techniques. Any optimization built into the Magic code is designed to improve efficiency of the Unix flat file storage system, not a database management system. The fact that the original Magic design does not take advantage of these fundamental concepts of the ObjectStore database management system may account for the failure of Magic to perform as well with ObjectStore as it does with its current Unix flat file management system. That the ObjectStore version of Magic performs best for hierarchy traversal demonstrates what Object Design considers to be the primary benefit of ObjectStore's architecture (9:61).

Overall, the performance results obtained from implementing Magic with ObjectStore are inconclusive. Benchmark tests indicate that ObjectStore performance falls within the ten percent increase criteria for only three of six areas. Actual usage of Magic, however, showed that the areas not meeting the performance criteria are unlikely to noticeably impact the overall performance of Magic. Modification of Magic's design to better take advantage of ObjectStore's database features would likely improve performance. Similarly, memory limitations of the existing implementation provided for a very unstable environ-

ment; however, proper implementation of transactions along with projected upgrades to the ObjectStore database system would likely eliminate this problem.

5.3.3 Conversion Cost Effectiveness. One of the primary costs of any software system is that necessary for software maintenance. These costs can be minimized if software changes affect only small, localized segments of code and if maintainers can easily understand the organization and functionality of existing code. Object-oriented computing attempts to minimize the impact of changes through data encapsulation, in which the underlying data is accessible only through a well-defined interface (19). Increased understanding is attained through abstraction, a concept in which the programmer's model of an object more closely approximates the user's conceptual model of the object (4). A database management system also supports data encapsulation and abstraction by providing mechanisms for defining storage structures and manipulating information (8).

ObjectStore supports both object-oriented computing and database management. With its persistent data structures and procedures for managing data, all input and output procedures can be eliminated from the program. This eliminates the complexity involved with transforming data from its flat file representation to the appropriate data structure in memory. Unfortunately, the implementation of Magic for this thesis does not take full advantage of ObjectStore's object-oriented computing facilities. Because the interface to the database is not well-defined, a change in the database structure may affect many segments of the system. By not converting Magic's existing *C* code to *C++* which is used by ObjectStore, additional complexity was added since both a *C* and a *C++* interface must be specified for each module.

ObjectStore provides the capability to greatly enhance the maintainability of Magic. Unfortunately, by failure to take full advantage of ObjectStore's object-oriented facilities, and through offsetting the maintenance advantages of a database management system with an extra interface for *C++*, the overall maintainability of Magic remains about the same. No maintenance cost savings are realized with the version of Magic implemented for this thesis.

On the other hand, the costs associated with converting to ObjectStore were relatively high. In four months of intense study and programming with ObjectStore, it was not possible to learn and understand every aspect of ObjectStore or to even completely understand any single aspect. No programming at all was accomplished using ObjectStore's collection and relation facilities. Versioning was not completely implemented due to faults in the ObjectStore system. In some instances, the documentation inadequately or improperly described ObjectStore functionality, thus requiring technical support from the designers of ObjectStore to resolve programming issues.

Another significant cost associated with conversion to ObjectStore was modifying *C* programs for compatibility with *C++*. This task could have been avoided by using ObjectStore's *C* library interface; however, this would increase the effort required to take advantage of the object-oriented programming facilities of *C++* at a later time.

The ObjectStore implementation of Magic for this thesis was not cost effective. The costs associated with learning the ObjectStore system and making *C* programs compatible with *C++* were not offset by any significant improvement in maintainability. A complete redesign using object-oriented techniques which take advantage of ObjectStore's data definition and manipulation language (DML) would significantly increase the understandability of the Magic code and likely reduce future maintenance costs. Such a redesign would also likely improve Magic's performance. The costs of a complete redesign would be high, however, suggesting that ObjectStore may be better suited for developing new systems or converting systems that are already object-oriented rather than converting a complex system design such as Magic's.

5.4 Recommendations for Further Research

This thesis did not take advantage of the object-oriented facilities of the ObjectStore DML, nor did it take full advantage of all the features of ObjectStore. While one of the goals of this thesis was to show a reduction in future maintenance costs, little was done toward attaining this goal. Object-oriented programming was expected to reduce maintenance costs; however, no such modifications were made to the Magic code. If the database manager module of Magic were truly object-oriented, the changes to implement

Magic with ObjectStore would have been limited to this module. To show that this is the case, future research should be directed toward implementing the database manager module of Magic using good object-oriented *C++* features. Creating a *C++ class* for the cell definition and each of its subordinate objects would significantly increase the understandability and maintainability of Magic's database manager.

As currently implemented with a single ObjectStore transaction, the ObjectStore version of Magic has even more severe memory limitations than the original version of Magic. The **drfmchip** circuit design is the largest circuit which can be loaded into memory. If transactions were limited to the smallest set of instructions necessary for maintaining database consistency, memory would only be limited by the amount of available disk space. Minimizing processing within a transaction would also improve the ability for concurrent access of the database by more than one user. Smaller transactions lock each database segment for less time, thereby giving other users quicker access to the same segment.

Two major features of ObjectStore were not used in this implementation of Magic: *relationships* and *collections*. Both have the potential to reduce the complexity of the Magic code. The cell definition structure includes relations to hash tables, cell labels, tiles, planes, and cell uses. Use of ObjectStore's relationship and *inverse relationship* features along with ObjectStore collections would potentially eliminate the complex list structures currently used for labels and cell uses. ObjectStore's facilities for traversing lists could be used instead, thus reducing the amount and complexity of Magic code. Magic's collection facilities could also directly replace the symbol tables used for cell definitions and cell configurations.

The versioning facility of ObjectStore was not completely implemented due to discrepancies with the ObjectStore system. When these discrepancies are fixed, versioning implementation should be completed to allow full functionality of the Magic system. In addition, versioning provides the capability for cooperative work on a circuit design in which two designers may work on the same circuit at the same time. If this is to be done, however, Magic must ensure that the two users do not make conflicting changes unless facilities are provided for managing these changes.

5.5 Summary

Overall performance of Magic as implemented for this thesis does not justify conversion of existing applications to use object-oriented databases. Many questions are left unanswered, however, due to the difficulties encountered while implementing Magic with ObjectStore. To answer these questions and take better advantage of ObjectStore's facilities, a number of proposals for additional research are presented. It is quite possible that complete implementation of these proposals would lead to a Magic system with significantly improved maintainability and performance. Object-oriented database management systems, while not fully proved in this thesis, still present the potential for greatly simplifying the development and maintenance of complex, data intensive, engineering applications.

Appendix A. *Raw Performance Test Results*

Original version of Magic using drfmchip

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
load drfmchip	0.06	0.180	4	7	2	0
	0.06	0.085	1	0	1	0
	0.06	0.078	1	1	1	1
	0.06	0.170	5	2	0	0
	0.08	0.120	2	2	1	0
	0.06	0.068	0	10	0	0
expand	6.71	10.18	124	833	115	3
	6.37	7.84	19	510	20	4
	6.76	9.83	121	745	123	3
	6.80	10.71	132	797	139	4
	6.67	9.88	122	850	129	3
	6.44	7.49	17	661	19	3
load test	0.01	0.012	0	16	0	0
	0.01	0.013	0	17	0	0
	0.01	0.012	0	16	0	0
	0.01	0.012	0	18	0	0
	0.01	0.012	0	16	0	0
	0.01	0.016	0	23	0	0
write test	0.01	0.147	0	17	1	7
	0.00	0.123	0	12	0	7
	0.04	0.093	0	12	0	6
	0.04	0.131	0	12	0	7
	0.02	0.091	0	12	0	6
	0.00	0.080	0	12	0	6
initialize	5.74	11.87	111	159	23	1
	5.58	10.90	85	202	5	2
	5.55	10.40	87	200	9	2
	5.58	11.16	95	196	16	1
	6.12	11.74	107	219	19	1
	5.45	8.18	7	245	8	1

Original version of Magic using drfmchip
getcell test nested 8 deep in subcell `mcell10`

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
writeall force	2.54	4.81	9	618	8	103
	1.61	3.28	2	88	0	90
	1.58	3.12	0	88	0	99
	2.38	3.93	0	88	3	94
	2.42	3.96	0	88	0	95
	1.61	3.26	0	89	5	90
getcell test	0.03	0.076	3	1	0	0
	0.02	0.013	0	0	0	0
	0.01	0.013	0	0	0	0
	0.02	0.085	2	0	0	0
	0.01	0.012	0	0	0	0
	0.01	0.010	0	1	0	0
drc catchup	96.24	97.59	26	93	0	0
	94.72	96.26	34	8	0	0
	94.73	95.70	13	13	0	0
	96.44	97.14	2	173	0	0
	94.88	95.55	0	2	0	0
	95.97	96.77	1	0	0	0

Original version of Magic using drfmchip
getcell test nested 4 deep in subcell `big_nandmux`

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
writeall force	0.70	1.47	0	48	0	45
	0.72	1.50	0	48	0	48
	0.68	1.48	0	48	0	45
	0.68	1.52	0	48	0	43
	0.66	1.51	0	48	0	50
	0.69	1.48	0	48	0	43
getcell test	0.01	0.0110	0	1	0	0
	0.00	0.0085	0	0	0	0
	0.00	0.0073	0	0	0	0
	0.00	0.0075	0	0	0	0
	0.00	0.0090	0	0	0	0
drc catchup	6.15	6.26	0	0	0	0
	6.20	6.23	0	0	0	0
	6.17	6.20	0	0	0	0
	6.18	6.22	0	0	0	0
	6.18	6.32	0	0	0	0
	6.22	6.27	0	0	0	0

ObjectStore (Cache Size 2048 sectors) version of Magic using drfmchip

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
load drfmchip	0.07	0.34	0	40	0	0
	0.04	0.36	0	39	0	0
	0.08	0.38	0	41	0	0
	0.09	0.41	0	45	0	0
	0.06	0.25	0	46	0	0
	0.07	0.38	0	45	0	0
expand	0.96	4.35	0	341	0	0
	1.09	5.52	0	363	0	0
	1.92	7.96	2	525	0	0
	1.95	6.47	1	525	0	0
	1.86	5.38	0	525	0	0
	1.86	5.78	0	525	0	0
load test	0.03	0.068	0	20	0	0
	0.02	0.107	0	21	0	0
	0.02	0.075	0	20	0	0
	0.02	0.022	0	20	0	0
	0.02	0.023	0	20	0	0
	0.02	0.025	0	20	0	0
initialize	5.69	12.77	36	475	9	1
	6.02	14.61	69	467	19	2
	5.92	16.23	94	459	17	2
	5.82	16.55	40	485	14	1
	6.00	18.23	36	515	9	2
	6.70	14.29	51	486	13	2

ObjectStore version of Magic using drfmchip
getcell test nested 8 deep in subcell `mcell10`

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
getcell test	0.44	0.54	2	15	0	0
	0.37	0.44	2	0	0	0
	0.49	0.49	0	160	0	0
	0.43	0.44	0	0	0	0
	0.38	0.38	0	1	0	0
	0.37	0.42	0	0	1	1
drc catchup	403.77	434.24	16	49	0	0
	366.54	369.31	0	713	0	0
	370.38	373.34	0	728	0	0
	360.74	363.55	0	14	0	0
	361.15	368.24	0	2	0	0
	349.10	351.85	0	1	0	0

ObjectStore version of Magic using drfmchip
getcell test nested 4 deep in subcell `big_nandmux`

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
getcell test	0.03	0.057	0	4	0	0
	0.02	0.024	0	0	0	0
	0.02	0.035	0	0	0	0
	0.02	0.015	0	0	0	0
	0.02	0.015	0	0	0	0
	0.02	0.015	0	0	0	0
drc catchup	18.43	19.58	11	307	0	0
	17.73	17.84	0	0	0	0
	18.34	18.45	0	2	0	0
	17.83	17.93	0	0	0	0
	18.53	18.72	0	0	0	0
	18.55	18.87	0	0	0	0

ObjectStore (Cache Size 640 sectors) version of Magic using drfmchip
getcell test nested 4 deep in subcell `big_nandmux`

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
load drfmchip	0.05	0.31	0	40	0	0
	0.03	0.38	0	51	0	0
	0.08	0.32	0	31	0	0
	0.04	0.29	0	29	0	0
	0.04	0.21	0	33	0	0
	0.04	0.48	0	33	0	0
expand	0.99	4.55	1	358	0	0
	1.06	1.06	0	0	0	0
	2.31	5.39	0	529	0	0
	2.15	5.79	0	528	0	0
	1.82	4.87	2	521	0	0
	0.94	4.59	0	345	0	0
load test	0.03	0.045	1	23	0	0
	0.02	0.050	0	23	0	0
	0.02	0.083	0	25	0	0
	0.02	0.023	0	23	0	0
	0.02	0.021	0	22	0	0
	0.02	0.021	0	22	0	0
initialize	5.81	13.83	5	527	4	1
	6.00	10.53	6	493	6	1
	6.01	15.19	81	434	11	2
	5.87	12.54	26	520	10	2
	5.72	21.45	3	491	4	1
	5.74	17.75	47	494	13	2
getcell test	0.09	0.400	4	31	0	0
	0.03	0.031	0	0	0	0
	0.03	0.030	0	0	0	0
drc catchup	23.97	24.86	13	118	0	0
	23.72	23.79	0	0	0	0
	23.34	23.58	0	0	0	0
	23.20	23.53	0	2	0	0
	23.41	23.80	0	1	0	0
	23.34	23.67	0	0	0	0

Original version of Magic using tut4a

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
load tut4a	0.00	0.210	4	0	1	0
	0.01	0.013	0	7	0	0
	0.01	0.013	0	7	0	0
	0.02	0.018	0	8	0	0
	0.01	0.013	0	7	0	0
	0.01	0.013	0	7	0	0
expand	0.05	0.190	5	0	2	0
	0.04	0.042	0	4	0	0
	0.03	0.034	0	2	0	0
	0.04	0.041	0	4	0	0
	0.06	0.100	0	4	0	0
	0.03	0.066	0	4	0	0
load test	0.01	0.0088	0	6	0	0
	0.01	0.0085	0	6	0	0
	0.01	0.0077	0	7	0	0
	0.00	0.0074	0	7	0	0
	0.00	0.0076	0	7	0	0
	0.01	0.0530	0	7	0	0
write test	0.00	0.149	0	17	1	7
	0.02	0.160	0	12	0	7
	0.01	0.157	0	12	0	7
	0.02	0.137	0	12	0	6
	0.01	0.128	0	12	0	6
	0.02	0.127	0	12	0	7
initialize	5.47	9.34	76	207	4	1
	5.47	8.10	4	314	4	2
	5.78	8.14	0	319	0	2
	5.94	8.52	0	319	0	0
	5.80	8.27	4	314	4	1
	5.50	8.85	0	319	0	1

Original version of Magic using tut4a

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
writeall force	0.07	0.553	1	26	0	23
	0.05	0.498	0	24	0	21
	0.05	0.479	0	24	0	21
	0.04	0.533	0	24	0	23
	0.10	0.507	0	24	0	21
	0.04	0.554	0	24	0	21
getcell test	0.00	0.0370	1	0	0	0
	0.00	0.0061	0	0	0	0
	0.01	0.0065	0	0	0	0
	0.00	0.0064	0	0	0	0
	0.00	0.0066	0	0	0	0
	0.00	0.0061	0	0	0	0
drc catchup	0.52	0.55	0	0	0	0
	0.55	0.56	0	0	0	0
	0.47	0.48	0	0	0	0
	0.55	0.55	0	0	0	0
	0.48	0.48	0	0	0	0
	0.55	0.56	0	0	0	0

ObjectStore (Cache Size 2048 sectors) version of Magic using tut4a

Command	CPU Time	Elapsed Time	Page Faults		Disk Blocks	
			with I/O	w/o I/O	In	Out
load tut4a	0.03	0.032	0	17	0	0
	0.02	0.036	0	22	0	0
	0.03	0.030	0	28	0	0
	0.03	0.034	0	32	0	0
	0.04	0.050	0	44	0	0
	0.03	0.038	0	39	0	0
expand	0.02	0.019	0	4	0	0
	0.02	0.021	0	6	0	0
	0.02	0.027	0	6	0	0
	0.02	0.028	0	7	0	0
	0.02	0.031	0	6	0	0
load test	0.01	0.015	0	14	0	0
	0.03	0.031	0	24	0	0
	0.01	0.015	0	13	0	0
	0.03	0.065	0	17	1	2
	0.01	0.016	0	14	0	0
	0.01	0.016	0	14	0	0
getcell test	0.03	0.099	2	4	0	0
	0.01	0.015	0	0	0	0
	0.01	0.015	0	0	0	0
	0.01	0.013	0	0	0	0
	0.01	0.013	0	0	0	0
	0.01	0.019	0	0	0	0
drc catchup	2.18	2.25	1	34	0	2
	1.83	1.86	0	0	0	0
	2.14	2.16	0	0	0	1
	1.87	1.89	0	0	0	0
	2.12	2.13	0	1	0	0
	1.88	1.89	0	0	0	0
initialize	5.64	9.71	4	507	6	3
	5.82	10.38	6	529	5	2
	5.73	9.99	4	510	4	1
	6.27	9.68	4	513	4	1
	5.74	10.00	5	512	4	1
	5.74	9.73	4	517	4	1

Bibliography

1. Berre, Arne J. and T. Lougenia Anderson. "The HyperModel Benchmark for Evaluating Object-Oriented Databases." In *Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD*, chapter 5, pages 75–91, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.
2. Cattel, R.G.G. "Object-Oriented DBMS Performance Measurement." In *Proceedings of the 2nd Workshop on OODBS*, pages 364–367, 1988.
3. Gupta, Rajiv and others. "An Object-Oriented VLSI CAD Framework," *Computer*, *22*:28–37 (May 1989).
4. Heiler, Sandra and others. "An Object-Oriented Approach to Data Management: Why Design Databases Need It." In *24th Design Automation Conference Proceedings*, pages 335–340, 1987.
5. Jacobs, Captain Timothy M. *OSmagic Programmers' Manual*. Air Force Institute of Technology, December 1991.
6. Jhingran, Anant and Michael Stonebraker. "Alternatives in Complex Object Representation: A Performance Perspective." In *Proceedings of the Sixth International Conference on Data Engineering*, pages 94–102, February 1990.
7. Kim, Won and others. "Indexing Techniques for Object-Oriented Databases." In *Object-Oriented Concepts, Databases, and Applications*, chapter 15, pages 371–394, New York: ACM Press, 1989.
8. Korth, H.F. and A. Silberschatz. *Database System Concepts*. New York: McGraw-Hill Book Company, 1986.
9. Lamb, Charles and others. "The ObjectStore Database System," *Communications of the ACM*, *34*:50–63 (October 1991).
10. Mayo, Robert N. and others. *1990 DECWRL/Livermore Magic Release*, 1990.
11. Object Design, Inc., Burlington, Massachusetts. *ObjectStore Administration and Development Tools*, March 1991.
12. Object Design, Inc., Burlington, Massachusetts. *ObjectStore Reference Manual*, March 1991.
13. Object Design, Inc., Burlington, Massachusetts. *ObjectStore User Guide*, March 1991.
14. Ousterhout, John K. *Magic Tutorial #6: Design-Rule Checking*. University of California, Berkeley, CA, 1990.
15. Ousterhout, John K. and others. "Magic: A VLSI Layout System." In *21st Design Automation Conference Proceedings*, pages 152–159, 1984.
16. Sawyer, Charlie and Steve Turner. Object Design, Inc. Technical Support, June - October 1991. Multiple telephone conversations.

17. Sidle, Thomas W. "Weaknesses of Commercial Data Base Management Systems in Engineering Applications." In *17th Design Automation Conference Proceedings*, pages 57–61, June 1980.

18. Stroustrup, Bjarne. *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1987.

19. Zdonik, Stanley B. and David Maier, editors. *Readings in Object-Oriented Database Systems*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.